MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 527

November 1979

The Dream of a Lifetime: A Lazy Scoping Mechanism

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Abstract:

We define a "rack", a data abstraction hybrid of a register and a stack. It is used for encapsulating the behavior of the kind of register whose contents may have an extent which requires that it be saved during the execution of an unknown piece of code. A rack can be implemented cleverly to achieve performance benefits over the usual implementation of a stack discipline. The basic idea is that we interpose a state machine controller between the rack abstraction and its stack/registers. This controller can act as an on-the-fly run-time peephole optimizer, eliding unnecessary stack operations.

We demonstrate the sorts of savings one might expect by using cleverly implemented racks in the context of a particular caller-saves implementation of an interpreter for the SCHEME dialect of LISP. For sample problems we can expect that only one out of every four pushes that would be done by a conventional machine will be done by the clever version.

Keywords: registers, stack discipline, stack architecture, register saving, procedure calling conventions, data abstraction

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by National Science Foundation Grant MCS77-04828 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

* Fannie and John Hertz Fellow

** Jolly Good Fellow

The Problem

We deal here with the problem of managing the use of a finite set of fast registers. Because the set of registers is finite and in fact usually much smaller than the total set of quantities of interest to the computation, the registers must be time-multiplexed, holding different quantities at different times. This leads immediately to the problem of how and when to move quantities to and from registers.

This problem is especially severe in the case of large systems of mutually recursive procedures. For small systems it may be possible to perform a complete analysis of all the procedures and their interactions and so find an optimal allocation of registers, but this is infeasible for large systems. Hence one usually adopts some standard method which always works but which may be overly conservative.

A typical technique is to use a push-down stack to preserve the contents of a register while it is being used to hold a new value for another purpose. The new value may in turn be preserved on the stack if a third purpose for that register arises. A value is saved by "pushing" it onto the stack at some time before the register is used for another purpose; it is restored by "popping" it back into the register after the new purpose has ended. The range of time during which a register is used for a given purpose (possibly interrupted by other, interpolated purposes) may be called an extent of a use the register (by analogy with the term when applied to the variables of a high-level programming language such as ALGOL). A stack can be used because a discipline is imposed requiring that the extents of quantities kept in a register be nested.

The nesting discipline is typically carried out by tying it to the nesting of procedure calls. There are two common conventions for using stacks in this context. In one, the "Caller Saves" convention, the calling procedure is responsible for pushing any values it will want later, and for retrieving them when the called procedure has returned. In this way a called procedure always has the full set of registers available for arbitrary use. In the other, the "Callee Saves" convention, the called procedure is responsible for preserving any registers that it uses. In this way a calling procedure can use any registers it pleases without worrying about called procedures destroying them.

There has been some debate as to which of these conventions is better. This debate has not been resolved because there is no good answer. Each convention is nonoptimal, and can perform much worse than the other for particular procedures. The Caller Saves convention is justly criticized because the caller may well save a register it did not have to (because, unknown to it, the called procedure does not use that register anyway). The Callee Saves convention is equally justly criticized because the called procedure may save the contents of a register which was not actually in use.

The problem of minimizing stack operations can be important because stack operations are usually significantly more expensive than register operations. This is true for typical computer systems using a hierarchy of memory devices; stacks, being of potentially unbounded size, are likely to overflow into slower memories. We are especially concerned with this problem in the context of hardware interpreters for high-level languages. (See [SCHEME Chip 0], [SCHEME Chip 1].) If the language allows closed procedures (FUNARGs) to be a first-class data type, which can be passed as an argument and returned as a value, and if we also allow control continuations to be similarly manipulated, then these problems are magnified because our stacks must be in garbage-collectable heap memory. (Cf. [SCHEME], [Revised Report], [Moses], [CONNIVER], and [Bobrow and Wegbreit].)

In this paper we will outline automatic methods of saving and restoring the values of registers which combine the good features of the classical conventions and which perform in every case as well as or better than either of the two classical conventions. We will demonstrate the effectiveness of our techniques for use of registers and stacks in the context of an interpreter for a dialect of LISP. We will compare the performance under various implementation choices. We will also describe how racks may be used in conventional computer architectures.

Other researchers have investigated and used techniques for improving the performance of stacks. [Burroughs] [LISP Machine] Such techniques generally implement modified push and pop operations which use registers to buffer the stack data. Our techniques are a generalization of those previously reported.

Although compiler optimizations can ameliorate the problem of optimal use of registers and stacks, the problem can not be fully resolved before run time, even if procedural arguments are disallowed (i.e. even if the targets of all procedure invocations are known at compile time). Which registers are in use and which are needed can depend on (a) which of several procedures is the caller, and (b) which of several data-dependent execution paths is taken within the called procedure. Hence in general not even the most complex optimizing compiler using either of the classical conventions can fully optimize register usage. If procedural arguments are to be implemented, then the problem is even more impossible.

Some negotiation must occur dynamically, at run time, between the various users of a register. Only if both the caller and the called procedure need to use a register should it be preserved for the caller's sake. More generally, the called procedure may not need a register, but one that it calls (or one indefinitely far down the call chain) may need the register. The negotiation protocol therefore cannot be just between caller and called procedure, but must extend over more then one level of call. The technique we present is actually independent of any procedure call mechanism; it

involves associating state information with each register describing whether or not it is in use.

The Strategy

By a "register" we mean something which can hold a single finite datum. There are two operations on registers: one can read the datum contained in the register, and one can store a new datum into it. The datum obtained by a read operation is always the datum most recently stored.

By a "stack" we mean something which can hold an indefinitely large number of finite data. There are two operations on stacks: one can push a new datum onto the stack, and one can pop a datum from the stack. The datum obtained by a pop operation is always the datum specified by the most recent matching push; that is, pushes and pops are "balanced" like parentheses.

The classical register-saving conventions deal with these two abstractions separately. A compiler typically issues read and store operations on registers for manipulating data, and issues push and pop operations for saving and restoring the registers.

When a stack is used to maintain the nested extents of a register it is not the actual pushing and popping which is of direct interest; it is the saving and restoring of the register. We introduce a composite abstraction (which we call a rack) which adds to the behavior of an ordinary register the ability to specify the beginning and ending of an extent (or, equivalently, to specify that the current value must be saved so as to be available later). Racks have four operations defined on them: fetch, assign, save, and restore. They are analogous to the read, store, push, and pop operations for a separate register and stack.

Our new abstraction is in fact to be implemented in terms of a stack and one or more registers. The save and restore operations appear to behave as push and pop stack operations, in that a restore operation causes to be available (to the fetch operation) that datum which was available before the corresponding save operation. However, performing a save does not necessarily (indeed, in some implementations never does!) perform a push on the internal stack; similarly, performing a restore does not necessarily perform a pop. Instead, pushes and pops are delayed until they are forced by subsequent operations. Hence a rack may be thought of as a kind of "lazy stack". In particular, a push or pop may occur during a fetch or assign operation.

We will present a series of implementations which embody different engineering tradeoffs. The implementations will be presented in approximately increasing order of complexity.

Rack Implementations

Each of the implementations given here embodies the same abstraction: a single register-cum-stack which responds to the four operations FETCH, ASSIGN, SAVE, and RESTORE. Each implementation will specify a set of internal registers and a stack, and procedures which implement the four operations in terms of READ and STORE operations on the internal registers and PUSH and POP operations on the internal stack.

The key idea is that each instance of a rack can have a state which encodes some of the history of previous operations. Each implementation is organized as a finite-state automaton which mediates between operation requests and the internal registers and stack. This automaton serves as an on-the-fly run-time peephole optimizer, which recognizes certain patterns of operations within a small window of events and transforms them so as to reduce the actual number of stack operations performed. Each rack has its own internal stack (as opposed to sharing one stack among several registers) so that the optimization can be performed independently on the operations to be performed on each rack. This will be important in minimizing the operations because otherwise operations on one rack could cause wasted operations on another. (We see this kind of inefficiency in systems which have "framed stacks" where each entry (a single frame) on the stack is a fixed pattern of saved state, much of which is irrelevant to the particular reason why that entry was constructed.)

We describe each implementation in the programming language SCHEME [Revised Report], a dialect of LISP which is lexically scoped and allows procedures to be passed as arguments and returned as values. (Appendix 2 contains some of the implementations expressed less abstractly and in an Algol-like language.) The use of procedural values allows us to describe a data abstraction as a procedural object which accepts and acts on messages sent to it. (Cf. [Actors] [Smalltalk].) Each implementation is a function which, when called, constructs and returns a fresh instance of the rack in the form of a closed procedure. The instance can be operated upon using the following operations which send appropriate messages to the instance:

```
(DEFINE (FETCH R) (R 'FETCH))

(DEFINE (ASSIGN R NEW-VALUE) ((R 'ASSIGN) NEW-VALUE))

(DEFINE (SAVE R) (R 'SAVE))

(DEFINE (RESTORE R) (R 'RESTORE))
```

That is, if the variable ENV has as its value an instance of the rack abstraction, then writing (FETCH ENV) will fetch the current contents of the rack, (ASSIGN ENV 3) will make the new contents of the current extent be 3, etc.

Our rack implementations will contain registers and stacks. A register or stack will also be modelled by closed procedures which take messages. A register can have a value stored in it or it can be read out:

```
(DEFINE (STORE REGISTER VALUE) ((REGISTER 'STORE) VALUE))

(DEFINE (READ REGISTER) (REGISTER 'READ))

(DEFINE (REGISTER)                                  ;Define a register to be an object
        (LET ((V NIL))                              ; which contains a quantity V, and
             (LAMBDA (OPERATION)                    ; responds to a request by dispatching
                     (CASEQ OPERATION               ; on the specified operation type.
                            ((READ) V)              ;For READ, return the quantity V.
                            ((STORE)                ;For STORE, return a function
                             (LAMBDA (NEWVALUE)     ; which will accept the new value
                                     (SETQ V NEWVALUE))))))))    ; and save it in V.
```

A stack is similarly behaviorally described:

```
(DEFINE (PUSH STACK TOP) ((STACK 'PUSH) TOP))

(DEFINE (POP STACK) (STACK 'POP))

(DEFINE (STACK)                                    ;Define a STACK to be an object
        (LET ((S NIL))                             ; which contains a list S (initially
             (LAMBDA (OPERATION)                   ; empty) and responds to a request
                     (CASEQ OPERATION              ; by dispatching.
                            ((PUSH)                ;For PUSH, return a function which
                             (LAMBDA (TOP)         ; will accept a new value and add it
                                     (SETQ S (CONS TOP S))))    ; to the front of the list S.
                            ((POP)                 ;For POP,
                             (IF (NOT (NULL S))    ; if the list S is not empty,
                                 (LET ((V (CAR S)))    ;  then save the first element,
                                      (SETQ S (CDR S)) ;  remove the first element from S,
                                      V)               ;  and return the former first element;
                                 (ERROR "Stack ran out - POP")))))))    ;ERROR if S is empty.
```

Ordinary Stack Implementation

As a short example of an implementation, we express the classical save-must-push/restore-must-pop convention. This is not very interesting except to demonstrate our notation and to serve as a benchmark for comparative performance measurements.

```
(DEFINE (STANDARD-STACK)
        (LET ((R (REGISTER)) (S (STACK)))
             (LAMBDA (OPERATION)
                     (CASEQ OPERATION
                            ((FETCH) (READ R))
                            ((ASSIGN)
                             (LAMBDA (NEWVALUE) (STORE R NEWVALUE)))
                            ((SAVE) (PUSH S (READ R)))
                            ((RESTORE) (STORE R (POP S)))))))
```

In this implementation, each instance creates two internal objects: a register R and a stack S. The current value is always kept in R. When a save operation is requested, the current value is always pushed onto S; during a restore operation, a pop always occurs.

Optimizing Pushes

The next implementation has two states, called AVAILABLE and IN-USE. The value associated with the current extent is always in the internal register R. The state encodes whether that value has been saved or not. That is, the state describes whether or not the internal register can be used freely or has been pressed into service as a virtual top-of-stack. The important idea here is that the state machine recognizes the operation sequence "SAVE; RESTORE" and treats it as a compound no-operation. It delays the pushing conceptually associated with a SAVE by moving to the state IN-USE. If the next operation is an ASSIGN (or another SAVE) then the push is performed after all. If the next operation is a RESTORE, however, then the state is simply reset to AVAILABLE, and a push and pop have been avoided.

```
(DEFINE (PUSH-OPTIMIZER)
        (LET ((R (REGISTER)) (STATE (REGISTER)) (S (STACK)))
             (STORE STATE 'AVAILABLE)   ;initial state
             (LAMBDA (OPERATION)
                     (CASEQ OPERATION
                            ((FETCH) (READ R))
                            ((ASSIGN)
                             (LAMBDA (NEWVALUE)
                                     (CASEQ (READ STATE)
                                            ((IN-USE)
                                             (PUSH S (READ R))
                                             (STORE STATE 'AVAILABLE)
                                             (STORE R NEWVALUE))
                                            ((AVAILABLE)
                                             (STORE R NEWVALUE)))))
                            ((SAVE)
                             (CASEQ (READ STATE)
                                    ((IN-USE) (PUSH S (READ R)))
                                    ((AVAILABLE) (STORE STATE 'IN-USE))))
                            ((RESTORE)
                             (CASEQ (READ STATE)
                                    ((IN-USE)
                                     (STORE STATE 'AVAILABLE))
                                    ((AVAILABLE) (STORE R (POP S)))))))))
```

Optimization of pushes can help considerably in a program which uses the Caller Saves convention. In this case, there are many calls to subprocedures which will not modify the registers of interest to the caller. The caller, however, will not in general know that these procedures are "safe". Even if we grant that the caller will

know which registers important procedures can potentially modify (a dangerous violation of modular organization, but one which is often made in highly optimized performance code), different subsets of the potentially modified registers will be actually modified depending on the arguments passed to the callee. Thus, our push optimizer can do better than an optimizing compiler under some circumstances. The reason is that it recognizes (temporal) sequences of relevant operations. Operations which do not affect the register of interest are not cluttering the view of our optimizer. Additionally, the optimizer can "see through" subroutine calls and other module boundaries.

Optimizing Pushes and Pops

The next implementation augments the previous implementation by recognizing some situations in which pops may be optimized as well as some pushes. It recognizes both "SAVE; RESTORE" and "RESTORE; SAVE" sequences and effectively elides them. The state machine has three states: IN-USE, AVAILABLE, and ON-STACK. The new state encodes where the value associated with the current extent is actually located: in the internal register or on the top of the internal stack. As before, if the value is not on the stack, it may either be protected (IN-USE) or the register may be available to accept a new value (AVAILABLE).

When does pop optimization buy anything? The state machine elides RESTORES followed by SAVES with no operations on that register in between. This is unlikely to happen in a well-written program using the Caller Saves convention because such sequences can be deleted by simple peephole optimization on the local data-flow of the program. However, in Callee Saves situations, it is often to be expected that two procedure calls occur in sequence, each calling a procedure which will clobber some particular register, which is not referenced by the code between the procedure calls. In this case such a useless sequence will occur. Again we see how this technique is very nice in that the code "removed" is not necessarily lexically adjacent, just adjacent logically in the flow of control.

```
(DEFINE (PUSH-AND-POP-OPTIMIZER)
        (LET ((R (REGISTER)) (STATE (REGISTER)) (S (STACK)))
             (STORE STATE 'AVAILABLE)   ;initial state
             (LAMBDA (OPERATION)
                     (CASEQ OPERATION
                            ((FETCH)
                             (CASEQ (READ STATE)
                                    ((IN-USE) (READ R))
                                    ((AVAILABLE) (READ R))
                                    ((ON-STACK)
                                     (STORE R (POP S))
                                     (STORE STATE 'AVAILABLE)
                                     (READ R))))
                            ((ASSIGN)
                             (LAMBDA (NEWVALUE)
                                     (CASEQ (READ STATE)
                                            ((IN-USE)
                                             (PUSH S (READ R))
                                             (STORE STATE 'AVAILABLE)
                                             (STORE R NEWVALUE))
                                            ((AVAILABLE)
                                             (STORE R NEWVALUE))
                                            ((ON-STACK)
                                             (POP S)
                                             (STORE STATE 'AVAILABLE)
                                             (STORE R NEWVALUE)))))
                            ((SAVE)
                             (CASEQ (READ STATE)
                                    ((IN-USE) (PUSH S (READ R)))
                                    ((AVAILABLE) (STORE STATE 'IN-USE))
                                    ((ON-STACK) (STORE STATE 'AVAILABLE))))
                            ((RESTORE)
                             (CASEQ (READ STATE)
                                    ((IN-USE) (STORE STATE 'AVAILABLE))
                                    ((AVAILABLE) (STORE STATE 'ON-STACK))
                                    ((ON-STACK) (POP S)))))))))
```

The important idea here is that the state machine recognizes the operation sequence "RESTORE; SAVE" and treats it as a compound no-operation. That is, it delays the popping conceptually associated with a RESTORE by moving to the state ON-STACK. If the next operation is a FETCH or ASSIGN (or another RESTORE) then the pop is performed after all. If the next operation is a SAVE, however, the state machine recognizes that the

relevant value is already on the stack and so the rack reverts to state AVAILABLE.

Now we have to be a little careful here. In describing the rack abstraction we never said whether a FETCH operation immediately after a SAVE operation is guaranteed to return the same value it would have before the SAVE was performed. If an implementation preserves value over SAVES we will say that it "duplicates" the extent before saving it. The simple stack implementation and the push-optimizer are both duplicating implementations. The push-and-pop-optimizer implementation would appear, at first, to do no damage to the value in the register during a SAVE operation, but consider the following sequence of operations:

```
(ASSIGN FOO 1)          ;this leaves the rack in state AVAILABLE (R is 1)
(SAVE FOO)              ;this leaves it IN-USE (R is 1)
(ASSIGN FOO 2)          ;this leaves it AVAILABLE (R is 2; 1 has been pushed)
(RESTORE FOO)           ;this leaves it ON-STACK (R is 2)
                        ;[doing a FETCH at this point would return 1, but also
                        ; would pop the stack and change the state to AVAILABLE]
(SAVE FOO)              ;this leaves it AVAILABLE (R is 2)
(FETCH FOO)             ;this returns 2
```

In a duplicating implementation, the result must be a 1, but in the implementation shown, the result will be 2! The duplication is an extra property which is needed in some applications and not in others.

We might think that we could fix up the push-and-pop-optimizer implementation so that it duplicated extents by replacing the following code in the case for SAVE:

```
((ON-STACK) (STORE STATE 'AVAILABLE))
```

with the somewhat more complex:

```
((ON-STACK)
 (STORE R (POP S))
 (STORE STATE 'IN-USE))
```

This indeed makes the rack duplicate, but it kills the pop optimization. It is not usually necessary for a rack to have a duplicating save. However, sometimes we need this operation explicitly. In this case we may find it necessary to implement a separate DUPLICATE operation which is the same as SAVE in most rack implementations and has the complex code above in this rack implementation.

Two-Cell Top-of-Stack Buffer

This implementation is presented for comparison. It is similar in spirit to the top-of-stack buffer used in the B6500/B7500 series computers [Burroughs]. There are two internal registers, R1 and R2, which buffer the stack operations. The value associated with the current extent is always in one of these two registers.; there are two states IN-R1 and IN-R2 indicating which is the case. An important feature of this implementation is that FETCH and ASSIGN never perform push or pop operations; only SAVE ever pushes, and only RESTORE ever pops. The registers provide a sliding window, however, within which pushes and pops may be elided.

```
(DEFINE (PDL-BUFFER)
  (LET ((R1 (REGISTER)) (R2 (REGISTER)) (STATE (REGISTER)) (S (STACK)))
    (STORE STATE 'IN-R2)        ;initial state.
    (LAMBDA (OPERATION)
      (CASEQ OPERATION
        ((FETCH)
         (CASEQ (READ STATE)
           ((IN-R1) (READ R1))
           ((IN-R2) (READ R2))))
        ((ASSIGN)
         (LAMBDA (NEWVALUE)
           (CASEQ (READ STATE)
             ((IN-R1)
              (STORE R1 NEWVALUE))
             ((IN-R2)
              (STORE R2 NEWVALUE)))))
        ((SAVE)
         (CASEQ (READ STATE)
           ((IN-R1)
            (PUSH S (READ R2))
            (STORE R2 (READ R1)))
           ((IN-R2)
            (STORE R1 (READ R2))
            (STORE STATE 'IN-R1))))
        ((RESTORE)
         (CASEQ (READ STATE)
           ((IN-R1) (STORE STATE 'IN-R2))
           ((IN-R2) (STORE R2 (POP S)))))))))
```

This rack implementation is duplicating and optimizes both pushes and pops. On the other hand, it needs two registers per rack (besides the stack-pointer register).

(On the third hand, it permits another kind of optimization not supported by any of the previous strategies: a sequence equivalent to "RESTORE; RESTORE; SAVE" can be performed without any popping if the rack had been in state IN-R1. Such a compound operation is typical of a binary arithmetic operation on a stack machine (such as are the B6500/B7500 series); addition, for example, is performed by effectively popping the top two elements of the stack and pushing back their sum in one operation. In state IN-R1 this is done by placing the sum of R1 and R2 in R2 and changing the state to IN-R2. If such binary operations are interleaved with operand fetches, then much stack arithmetic can be performed with no actual pushes or pops.)

Collapsing Extents Using a Counter

In some applications the values held in a register during successive nested extents often are identical, and change only infrequently. Consider, for example, a situation where the register is used to hold a parameter passed down from one level to another, from one procedure to another, without change. The idea here is that if several consecutive values are the same, this fact can be encoded by keeping only one copy of the value plus a count of the replications. (We will exhibit an application below where this idea yields a substantial performance improvement.) One way to think about this is to consider the stack to be "run-length encoded". Another way is to consider the value to have a "protection count", somewhat like a reference count as used in storage allocators [Weizenbaum] [SLIP]; thus the same value can be "IN-USE" more than once.

This is not without associated overhead. When a value is eventually actually pushed onto the stack, two items must be pushed: the value and the count. Hence if in practice consecutive values are not often identical, then this implementation performs twice as many stack operations as previous implementation we have presented (for example PUSH-OPTIMIZER). There is also a complexity overhead: though there are no explicit states (the value of the count contains the state), the finite-state machine which mediates between user and actual stack must be capable of doing simple arithmetic (adding and subtracting one); this may be important when implementing this technique in hardware.

```
(DEFINE (PUSH-COUNTER NAME)
        (LET ((R (REGISTER NAME)) (COUNT (REGISTER NAME)) (S (STACK NAME)))
             (STORE COUNT 0)    ;Initialization.
             (LAMBDA (OPERATION)
                     (CASEQ OPERATION
                            ((FETCH) (READ R))
                            ((ASSIGN)
                             (LAMBDA (NEWVALUE)
                                (BLOCK (IF (NOT (= 0 (READ COUNT)))
                                           (BLOCK (PUSH S (READ R))
                                                  (PUSH S (- (READ COUNT) 1))
                                                  (STORE COUNT 0)))
                                       (STORE R NEWVALUE))))
                            ((SAVE) (STORE COUNT (+ 1 (READ COUNT))))
                            ((RESTORE)
                             (IF (= (READ COUNT) 0)
                                 (BLOCK (STORE COUNT (POP S))
                                        (STORE R (POP S)))
                                 (STORE COUNT (- (READ COUNT) 1)))))))))
```

The critical idea is the code for ASSIGN. If the count is non-zero, then the current value is serving for more than one nested extent (just as in previous implementations state IN-USE implied that the value served for two extents). In this case one extent must be de-collapsed from the others so that the assignment may be performed. (A very tricky implementation might first check to see whether the new assigned value were the same as the old one or the previous one, and attempt to collapse extents! It is not at all clear that this is worthwhile.) This push-counter implementation of racks has a duplicating SAVE operation (after all, the whole point is to use this implementation in situations where consecutive extents are often the same, and this probably occurs by algorithmic duplication rather than computational accident.)

The Impact of Racks on the Performance of an Evaluator for SCHEME

The proof of the pudding is in the eating. We have tried the idea of using racks to implement the saving and restoring of registers in a caller-saves implementation of an interpretive evaluator for (a dialect of) the SCHEME language (see Appendix 1 for a complete listing of the evaluator.) This provides a non-trivial exercise of register saving because it performs a highly recursive operation whose precise actions depend in a complex way on the data being processed (i.e. the SCHEME program being interpreted). We ran the interpreter on a set of test problems with those registers which are saved implemented as various kinds of racks.

In the particular interpreter we used there are five registers which are saved at various points in the interpreter. They are: RETPC, ENV, UNEV, ARGL, FUN.

RETPC is used to hold the "return address" for an evaluation of a subexpression. The interpreter may recurse to evaluate a subexpression for one of a number of reasons. RETPC holds the reason for the current subevaluation. For example, it may be that the subexpression is an argument to a procedure call. In this case the evaluator will have to evaluate the following s (if any) and then apply the procedure. The subexpression may be the predicate part of a conditional, in which case the evaluator wants to use the value of this expression to decide whether to proceed with the consequent of the conditional or the alternative of the conditional.

ENV is used to hold the environment which is the map of identifiers to their values. Since SCHEME is lexically scoped, this map is never required at the moment a procedure is called because the procedure is enclosed within its own lexical environment (that is, the procedure object contains an environment in which to execute the code for the procedure). The actual new environment is made by binding the formal parameters of the procedure to the (evaluated) actual parameters of the call, and adding these bindings to the closure environment. The new environment is assigned to the environment register and execution begins on the body of the procedure. Thus, it is necessary to preserve the environment of an expression across a procedure call if more computations are to be done with that environment after the procedure returns. This may happen if the subexpression is the predicate of a conditional, or is an argument in a procedure call expression after which other arguments must be evaluated before the procedure can be called.

UNEV is a register which is used to remember the unevaluated part of an expression across the evaluation of a subexpression. Thus, UNEV is used to hold the consequent and the alternative in a conditional and the rest of the unevaluated arguments in a call.

ARGL is used to hold the list of already evaluated arguments for a procedure while the next argument is being evaluated. It eventually contains the entire list of arguments and is used to construct the environment when binding the formal parameters to the actual parameters (stored in ARGL) at the procedure invocation.

FUN holds the procedure to be invoked after all of its arguments are evaluated. It is necessary because the evaluator we are using evaluates an expression from left to right, and in that syntax the procedure comes first, followed by its arguments.

The evaluator we are using has other registers for temporary storage of values and expressions. Those registers are not interesting for our test as they are never saved.

To measure the performance of the evaluator we use the following doubly recursive method of computing Fibonacci numbers. This test is interesting in that it exercises the recursion mechanism of the interpreter rather thoroughly and is thus a good indicator of the overhead of evaluating subexpressions. The problem is to evaluate (FIB 4) where we define:

```
(DEFINE (FIB N)
        (COND ((= N 0) 0)
              ((= N 1) 1)
              (T (+ (FIB (- N 1))
                    (FIB (- N 2))))))
```

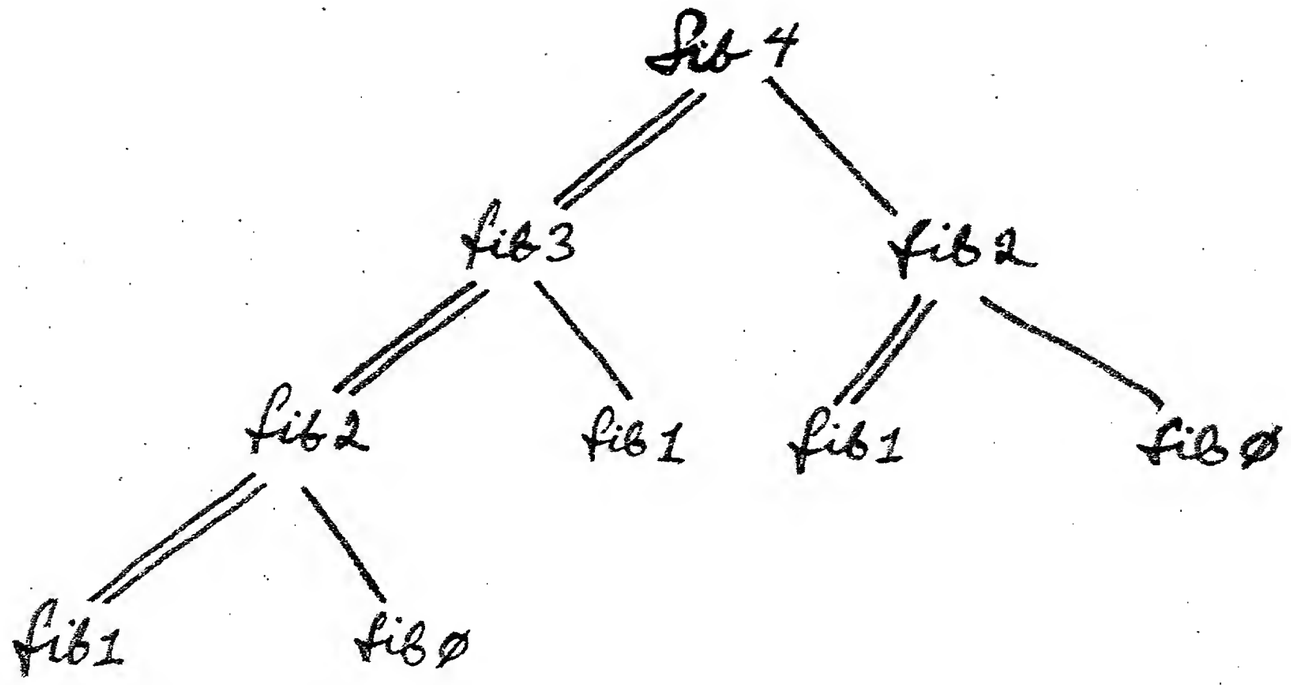
The results are summarized in the following table. For each implementation, for each register, we give the actual number of items pushed and popped in the execution of the test example.

	RETPC	ENV	UNEV	ARGL	FUN	Total
STANDARD-STACK	123	85	85	65	37	395
PUSH-OPTIMIZER	33	20	20	16	12	101
PUSH-AND-POP-OPTIMIZER	33	20	20	16	12	101
PDL-BUFFER	33	20	20	16	12	101
PUSH-COUNTER	66	8	40	32	24	170

We can see that the simple push-optimizer is a tremendous improvement over a simple unoptimizing stack. Additionally, in this caller-saves discipline, the push-and-pop optimizer gives us no advantage for its added complexity. The pdl-buffer, which requires an extra hardware register, makes no difference either. The push optimizer is a simple two-state machine so it is trivial to implement in hardware.

The push-counter is worse than the push-optimizer for every register except the environment register ENV. In that case, it makes a remarkable difference. What has happened is that the evaluator does not save the environment unless it is logically necessary to do so to allow the computation to proceed. The only reason that the environment is ever saved is because it will be needed after a recursive call to the evaluator (and this happens to occur only when evaluating an argument to a procedure (other than the last argument) or the predicate of a conditional; because SCHEME is lexically scoped, the environment is not needed to apply a procedure (in contrast to the implementation of LISP 1.5 and its successors), because each procedure is enclosed with its own favorite environment). Thus, in the Fibonacci evaluation the only reason to save the environment is when the first argument of + is evaluated because the evaluation of the second argument will need that environment. The environment will be assigned (and therefore pushed) upon applying the subcall to FIB so it must be saved.

Thus we see that the environment must actually be pushed precisely 4 times:



The Computation Tree for (FIB 4)
The doubled edges show recursions over which the environment must be saved.

From this data, we infer that it is best, for this particular program (the interpreter), to let each rack be implemented by PUSH-OPTIMIZER, except for ENV, which should be implemented by PUSH-COUNTER. Let us call that implementation the OPTIMAL strategy for implementing the interpreter. We will now compare the gains of using OPTIMAL racks over STANDARD-STACK racks. For the problem (FIB 4) the OPTIMAL strategy uses only 23% of the pushes required by the STANDARD-STACK strategy:

(FIB 4)	RETPC	ENV	UNEV	ARGL	FUN	Total	
STANDARD-STACK	123	85	85	65	37	395	
OPTIMAL	33	8	20	16	12	89	and 89/395=.225...

Let us see how this varies with the argument. The following is the data for (FIB 5).

(FIB 5)	RETPC	ENV	UNEV	ARGL	FUN	Total	
STANDARD-STACK	209	145	145	111	63	673	
OPTIMAL	56	14	34	28	21	153	and 153/673=.227...

It still seems to be about 23%, independent of the input argument! This should not be too surprising. Giving a larger argument to FIB merely causes the same code to be executed more times. The use of racks does not optimize the algorithm being interpreted; it merely gains a constant factor of speed for the interpreter. Now it gains different constant factors for different parts of the interpreter, so the speed-up factor may be different for interpreting a program other than FIB. For example, if we define factorial by the standard singly recursive definition:

```
(DEFINE (FACT N)
        (COND ((= N 0) 1)
              (T (* N (FACT (- N 1))))))
```

then the savings on (FACT 4) are more substantial.

(FACT 4)	RETPC	ENV	UNEV	ARGL	FUN	Total	
STANDARD-STACK	59	40	40	31	18	188	
OPTIMAL	14	0	5	8	8	35	and 35/188=.186...

The optimal strategy does only 19% as many pushes as the standard stack implementation. This figure remains the same for (FACT 5):

(FACT 5)	RETPC	ENV	UNEV	ARGL	FUN	Total	
STANDARD-STACK	72	49	49	38	22	230	
OPTIMAL	17	0	6	10	10	43	and 43/230=.186...

Note that there are precisely zero environment pushes required to compute factorials. This optimization is, however very sensitive to the exact form of the code being interpreted (and thus to the particular execution paths taken within the interpreter). If we instead defined factorial with the order of arguments to * reversed:

```
(DEFINE (FACT1 N)
        (COND ((= N 0) 1)
              (T (* (FACT1 (- N 1)) N))))
```

it would be necessary to save the environment for each recursive call to fact because the environment will be needed after the recursive evaluation to access the value of n.

For an iterative implementation of factorial the savings are even greater:

```
(DEFINE (FACT2 N) (FACT-ITER (- N 1) N))

(DEFINE (FACT-ITER COUNT ANS)
        (COND ((= COUNT 0) ANS)
              (T (FACT-ITER (- COUNT 1) (* COUNT ANS)))))
```

For (FACT2 4) the figures are:

(FACT2 4)	RETPC	ENV	UNEV	ARGL	FUN	Total	
STANDARD-STACK	55	38	38	31	16	178	
OPTIMAL	12	0	8	7	4	31	and 31/178=.174...

Here the OPTIMAL strategy requires only about 17% of the pushes required by the STANDARD-STACK strategy.

Use of Racks in "Standard" Computer Architectures

Racks could enhance the performance of a computer built around a standard general-register-with-stacks architecture (such as the PDP-11, PDP-10, 8080, Z-8000, etc.). A general purpose rack register A would be implemented in hardware as two registers (one (A.R) large enough to hold a datum, the other (A.S) large enough to hold an address (the stack pointer)), and one or two status bits or a counter. Any instruction which reads a register would read it by performing the mechanics of the rack FETCH operation; any instruction which stores into a register would perform the ASSIGN operation.

The push-optimizer and push-counter methods are especially attractive here because the FETCH operation merely reads the datum from the datum register A.R, and so there is no overhead for reading a register. Assigning to a register might perform a push, however; thus a PDP-11 instruction MOV R0,R1 might perform a memory operation to push the old contents of R1 before copying the contents of R0 into it. This would occur only if there had been a previous request to SAVE register R1 which had been delayed. In any case, referring to R0 could not cause a memory operation.

It would be desirable to provide special instructions to SAVE and RESTORE registers. These would be analogous to PUSH and POP operations as usually used. One difference is that a SAVE operation merely guarantees to preserve a register; it does not guarantee to put the datum in memory on the stack right away. Hence code which expects to examine a saved value on the stack cannot simply read the stack pointer and index off of it. On the other hand, an additional operation can be provided to read the stack pointer after performing any delayed operations. Thus, for the push-counter strategy, such an operation would first check the state; if IN-USE, the datum in the register would be pushed and the state reset to AVAILABLE. Then the (new) stack pointer could be returned. For the special case of indexing into the stack (to read or write an entry), a special addressing mode could be provided which would index off a register's stack pointer and take the status bit into consideration so as to fetch the expected element as if a SAVE operation always performed a PUSH. That is, if the state were AVAILABLE, then indexing would be normal; while if it were IN-USE, then indexing by zero would fetch the register datum, while other index offsets would have to be adjusted by one. In this way the push which might be performed by merely reading the stack pointer can be avoided (because it is known that the value of the stack pointer is to be used in a limited way).

Some existing architectures already provide special instructions to push more than one register onto a stack. These could be adapted to perform SAVE operations instead. Others automatically push registers onto a stack when a procedure call occurs; on the VAX, for example, a two-byte bit mask preceding the first instruction of a procedure specifies which registers to push. If SAVE operations were done instead, then the actual number of pushes could be reduced. (Because this is an example of the Callee Saves convention, the push-and-pop-optimizer implementation might be appropriate here.)

Racks seem to be most effective on code which uses recursion heavily and which follows unpredictable paths through the code of each procedure, such that different paths use different registers. LISP code tends to be written in this style: each procedure immediately does a multi-way branch by testing the arguments, and then performs a computation, possibly by recursion, of greater or lesser complexity. We speculate that many programs coded in one or another "structured" style with heavy use of procedure calls will exhibit these characteristics and so be aided by an implementation which uses racks to optimize register saves.

Appendix 2 contains pseudo-Algol code (essentially equivalent to the LISP code given above) describing how the rack operations and stack indexing might be implemented in hardware. In real machines it is sometimes necessary to be able to get a control structure summary for interrupts, process switching, and non-standard exit conditions. The use of cleverly implemented racks may increase the cost of these operations because some of the state of the machine is distributed into the states of the rack abstractions. Thus it is necessary to be able to dump the state of a rack and restore it easily. This can be done quite painlessly in the cases we have illustrated and the code in Appendix 2 shows procedures for obtaining such stack pointers when necessary.

Conclusions

We have defined a data abstraction which we call a "rack" which may be thought of as a hybrid of a register and a stack. It is useful for encapsulating the behavior of the kind of register whose contents may have an extent which requires that it be saved during the execution of an unknown piece of code.

A rack can be implemented in many ways, the simplest being just a register which is saved on a stack in the usual way, but with other choices of implementation leading to increased efficiency (if we assume that stack accesses are expensive by comparison to register accesses). The basic idea is that we interpose a state machine controller between the rack abstraction and its stack/registers. This controller can act

as an on-the-fly run-time peephole optimizer, eliding unnecessary stack operations.

Each of the implementations we have exhibited has different virtues. The push-optimizer implementation is simple, requiring only a single state bit, and works well for code which uses a Caller Saves convention. The push-and-pop-optimizer also works for a Callee Saves convention. The pdl-buffer implementation requires one more register than push-optimizer, but works just as well, and in addition supports binary stack-arithmetic operators well. The push-counter implementation requires a counter rather than a one- or two-bit state, but run-length-encodes the stack, which can substantially improve performance if nested extents often have identical values.

There are many other possible implementations of racks. A rack is an abstract data structure. Just as a set may be implemented as a linked list of elements, a bit string with 1-bits indicating contained elements, or a membership predicate, so a rack may be implemented in many ways, which will have different performance characteristics under various conditions of use.

We have demonstrated the sorts of savings one might expect by using cleverly implemented racks. On a particular caller-saves implementation of an interpreter for the SCHEME dialect of LISP, we have seen that if push-optimization (a simple 2-state machine) is used on all registers except the environment register and if a push-counter is used on the environment register, then for sample problems we can expect that typically only one out of every four pushes that would be done by a conventional machine will be done by the clever version. Indeed, this can be very significant if the stacks are expensive. (This is the case in the MIT-AI/XEROX-PARC SCHEME-79 single-chip LISP interpreter [SCHEME Chip 2], a VLSI microprocesssor which directly interprets LISP code, automatically manages storage as a garbage-collected heap, and keeps its stacks in the heap. Dealing with external memory is much slower than manipulating an on-chip state bit. We intend to design a version of this chip which uses racks to improve performance (indeed, the notion of a rack as a generalized data abstraction was developed in an attempt to optimize earlier versions of the chip.) We expect that the savings can become even larger with slightly different designs for our interpreter. For example, if we use a number of separate racks to hold arguments for specific primitive operators, rather than collecting a list of them in a single rack ARGL, then performance may be even further improved.)

Appendix 1
The Test Interpreter

The following is a listing of the interpreter we used for developing the results of the tests we have displayed, using racks to implement the saveable registers. The interpreter evaluates expressions written in (a subset of) SCHEME [Revised Report]. The environment register, ENV, is the only one which demands a duplicating save; this is indicated in the code by using the DUPLICATE operator rather than SAVE.

While the code given here for the interpreter is itself written in SCHEME, it is so coded that it does not use the variable-binding and recursive procedure-call mechanisms of SCHEME. Instead, racks are used to save and restore items in a stack-like discipline; the items so saved include both data objects and "return addresses". In effect, the recursive evaluation algorithm is implemented in terms of an explicit stack (i.e. a rack), just as it would be in an assembly language.

```
(DEFINE (EVAL-EXP-RESULT-TO PC)
        (ASSIGN RETPC PC)
        (SAVE RETPC)
        (EVAL-DISPATCH))

(DEFINE (POPJ)
        (RESTORE RETPC)
        ((FETCH RETPC)))
```

```
(DEFINE (EVAL-DISPATCH)
        (COND ((ATOM (FETCH FORM))
               (COND ((NUMBERP (FETCH FORM))
                      (ASSIGN VAL (FETCH FORM))
                      (POPJ))
                     (T
                      (ASSIGN VAL (VALUE (FETCH FORM) (FETCH ENV)))
                      (POPJ))))
              ((EQ (CAR (FETCH FORM)) 'QUOTE)
               (ASSIGN VAL (CADR (FETCH FORM)))
               (POPJ))
              ((EQ (CAR (FETCH FORM)) 'LAMBDA)
               (ASSIGN VAL
                       (LIST '&PROCEDURE
                             (CADR (FETCH FORM))
                             (CADDR (FETCH FORM))
                             (FETCH ENV)))
               (POPJ))
              ((EQ (CAR (FETCH FORM)) 'COND)
               (ASSIGN UNEV (FETCH FORM))
               (EVCOND-PRED))
              ((NULL (CDR (FETCH FORM)))
               (ASSIGN FORM (CAR (FETCH FORM)))
               (EVAL-EXP-RESULT-TO APPLY-NO-ARGS))
              (T
               (ASSIGN UNEV (FETCH FORM))
               (ASSIGN FORM (CAR (FETCH FORM)))
               (DUPLICATE ENV)
               (SAVE UNEV)
               (EVAL-EXP-RESULT-TO EVAL-ARGS))))

(DEFINE (EVCOND-PRED)
        (ASSIGN UNEV (CDR (FETCH UNEV)))
        (ASSIGN FORM (CAAR (FETCH UNEV)))
        (DUPLICATE ENV)
        (SAVE UNEV)
        (EVAL-EXP-RESULT-TO EVCOND-DECIDE))
```

```
(DEFINE (EVCOND-DECIDE)
        (COND ((FETCH VAL)
               (RESTORE ENV)
               (RESTORE UNEV)
               (ASSIGN FORM (CADAR (FETCH UNEV)))
               (EVAL-DISPATCH))
              (T
               (RESTORE UNEV)
               (RESTORE ENV)
               (EVCOND-PRED))))

(DEFINE (APPLY-NO-ARGS)
        (ASSIGN FUN (FETCH VAL))
        (ASSIGN ARGL NIL)
        (INTERNAL-APPLY))

(DEFINE (EVAL-ARGS)
        (ASSIGN FUN (FETCH VAL))
        (SAVE FUN)
        (ASSIGN ARGL NIL)
        (EVAL-ARGS1))

(DEFINE (EVAL-ARGS1)
        (SAVE ARGL)
        (RESTORE UNEV)
        (ASSIGN UNEV (CDR (FETCH UNEV)))
        (ASSIGN FORM (CAR (FETCH UNEV)))
        (RESTORE ENV)
        (COND ((NULL (CDR (FETCH UNEV)))
               (EVAL-EXP-RESULT-TO EVAL-LAST-ARG))
              (T
               (DUPLICATE ENV)
               (SAVE UNEV)
               (EVAL-EXP-RESULT-TO EVAL-ARGS2))))

(DEFINE (EVAL-ARGS2)
        (RESTORE ARGL)
        (ASSIGN ARGL (CONS (FETCH VAL) (FETCH ARGL)))
        (EVAL-ARGS1))
```

```
(DEFINE (EVAL-LAST-ARG)
        (RESTORE FUN)
        (RESTORE ARGL)
        (ASSIGN ARGL (CONS (FETCH VAL) (FETCH ARGL)))
        (INTERNAL-APPLY))

(DEFINE (INTERNAL-APPLY)
        (COND ((PRIMOP? (FETCH FUN))
               (ASSIGN VAL (PRIMOP-APPLY (FETCH FUN) (FETCH ARGL)))
               (POPJ))
              ((EQ (CAR (FETCH FUN)) '&PROCEDURE)
               (ASSIGN ENV
                       (BIND (CADR (FETCH FUN))
                             (FETCH ARGL)
                             (CADDDR (FETCH FUN))))
               (ASSIGN FORM (CADDR (FETCH FUN)))
               (EVAL-DISPATCH))
              (T (BREAK |UNKNOWN FUNCTION TYPE|))))
```

Appendix 2
Rack Instructions

Here we present pseudo-Algol code for two implementations of racks (push-optimize and push-counter) in a style suited for use in standard computer architectures. This code is essentially equivalent to the LISP code presented in the main text; the LISP code, however, assumes that stacks and integer counters may grow indefinitely in size. The code given here explicitly indicates memory references and checks for stack overflow and arithmetic overflow. Another difference is that the LISP code for racks uses an abstract interface to the stack data type, while the code here explicitly implements a stack stored linearly in memory, growing upwards (as on a PDP-10, and unlike a PDP-11 or VAX, for example).

Let the array MEMORY[] represent the primary memory in which stack data is stored. We present the two implementations in parallel: push-optimize on the left, and push-counter on the right. To make the code align nicely for comparison, blank lines are intentionally inserted in the code.

A REGISTER implemented as a rack is a structure containing three registers. One, R, holds a data word; one, S (the stack pointer), holds an address. For the push-optimizer implementation the third one is a bit; for the push-counter implementation

the third one is a counter, which may be of any size. Ideally the counter should be large enough to count most runs of identically-valued nested extents, but there is a trade-off here: the smaller the counter, the more often pushes must be done when it overflows, but the larger it is, the more bits must be pushed when a push does occur. (We suspect that in practice the size of the counter will probably be determined by other architectural constraints.)

```
/* PUSH-OPTIMIZE IMPLEMENTATION */
TYPE REGISTER =
   RECORD(R: DATAWORD,
          S: ADDRESS,
          AVAIL: BIT);
```

```
/* PUSH-COUNTER IMPLEMENTATION */
TYPE REGISTER =
   RECORD(R: DATAWORD,
          S: ADDRESS,
          COUNT: WORD);
```

Reading a register (implemented as a rack, of course) is straightforward.

```
FUNCTION FETCH(REG)
   RETURN(REG.R);
```

```
FUNCTION FETCH(REG)
   RETURN(REG.R);
```

Assignment is less straightforward...

```
PROCEDURE ASSIGN(REG, NEWVALUE)
   BEGIN
      IF NOT REG.AVAIL THEN
         BEGIN
            REG.S := REG.S + 1;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S] := REG.R;

            REG.AVAIL := TRUE;
         END;
      REG.R := NEWVALUE;
   END;
```

```
PROCEDURE ASSIGN(REG, NEWVALUE)
   BEGIN
      IF REG.COUNT > 0 THEN
         BEGIN
            REG.S := REG.S + 2;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S-1] := REG.R;
            MEMORY[REG.S] := REG.COUNT - 1;
            REG.COUNT := 0;
         END;
      REG.R := NEWVALUE;
   END;
```

The entire body of either version of ASSIGN could be preceded by the conditional test IF NOT (NEWVALUE = REG.R) THEN. If this test can be performed cheaply and if it is likely that a value might be assigned which is already the value of the current extent of the register, then this test may be worthwhile, as it may avoid some pushes.

In the push-counter implementation of SAVE, we must check for overflow of the counter. If this occurs, the old count is pushed onto the stack and the counter reset.

```
PROCEDURE SAVE(REG)
   IF REG.AVAIL
      THEN REG.AVAIL := FALSE
      ELSE BEGIN
              REG.S := REG.S + 1;
              CHECK_STACK_OVERFLOW(REG.S);
              MEMORY[REG.S] := REG.R;


           END;
```

```
PROCEDURE SAVE(REG)
   IF REG.COUNT < MAX_WORD_VALUE
      THEN REG.COUNT := REG.COUNT + 1
      ELSE BEGIN
              REG.S := REG.S + 2;
              CHECK_STACK_OVERFLOW(REG.S);
              MEMORY[REG.S-1] := REG.R;
              MEMORY[REG.S] := REG.COUNT;
              REG.COUNT := 0;
           END;
```

If the result of a counter incrementation is zero when the counter overflows, then the push-counter SAVE may be simplified to:

```
PROCEDURE SAVE(REG)
   BEGIN
      REG.COUNT := REG.COUNT + 1
      IF REG.COUNT = 0 THEN
         BEGIN
            REG.S := REG.S + 2;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S-1] := REG.R;
            MEMORY[REG.S] := REG.COUNT;
         END;
   END;
```

In RESTORE we must check for stack underflow (trying to pop too many entries off the stack). This can occur if more RESTORE operations are performed than SAVE operations.

```
PROCEDURE RESTORE(REG)
   IF REG.AVAIL
      THEN BEGIN
              CHECK_STACK_UNDERFLOW(REG.S);


              REG.R := MEMORY[REG.S];
              REG.S := REG.S - 1;
           END
      ELSE REG.AVAIL := TRUE;
```

```
PROCEDURE RESTORE(REG)
   IF COUNT = 0
      THEN BEGIN
              CHECK_STACK_UNDERFLOW(REG.S);
              REG.COUNT := MEMORY[REG.S];
              REG.R := MEMORY[REG.S-1];
              REG.S := REG.S - 2;
           END
      ELSE REG.COUNT := REG.COUNT - 1;
```

Function STACKPTR reads the stack pointer after forcing the rack into a known state, so that all saved values are on the stack; that is, any delayed push is performed first. Note how similar the code is to that for ASSIGN.

```
PROCEDURE STACKPTR(REG)
   BEGIN
      IF NOT REG.AVAIL THEN
         BEGIN
            REG.S := REG.S + 1;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S] := REG.R;

            REG.AVAIL := TRUE;
         END;
      RETURN(REG.S);
   END;
```

```
PROCEDURE STACKPTR(REG)
   BEGIN
      IF REG.COUNT > 0 THEN
         BEGIN
            REG.S := REG.S + 2;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S-1] := REG.R;
            MEMORY[REG.S] := REG.COUNT - 1;
            REG.COUNT := 0;
         END;
      RETURN(REG.S);
   END;
```

Function STACKINDEX logically indexes into the stack associated with a register. The argument N must be a non-positive offset into the stack. If N is zero, the most recently saved value is returned; if it is -1, the next most recently saved value; and so on. (It is possible to code an equivalent version for the push-counter implementation, but it is rather inefficient, as it must scan the stack linearly, summing the saved counts.)

```
FUNCTION STACKINDEX(REG, N)
   IF R.AVAIL
      THEN RETURN(MEMORY[REG.S+N])
   ELSE IF N = 0
      THEN RETURN(REG.R)
   ELSE RETURN(MEMORY[REG.S+N+1]);
```

Procedure STACKINDEXWRITE is similar, but takes a value to be written into the slot referred to be indexing. It would be very difficult to write a version of this for the push-counter strategy; hence if this operation is needed then that strategy should not be used.

```
PROCEDURE STACKINDEXWRITE(REG, N, NEWVALUE)
   IF R.AVAIL
      THEN MEMORY[REG.S+N] := NEWVALUE
   ELSE IF N = 0
      THEN BEGIN
            REG.S := REG.S + 1;
            CHECK_STACK_OVERFLOW(REG.S);
            MEMORY[REG.S] := NEWVALUE;
            REG.AVAIL := TRUE;
         END;
   ELSE MEMORY[REG.S+N+1] := NEWVALUE;
```

Historical Note

The basic idea for a rack came suddenly to one of the authors (Sussman) in a dream at three o'clock in the morning. He had been worrying about the lifetimes of quantities saved on the stack in a LISP interpreter. From this is derived the title of this report. Also, we have come to refer informally to the general technique of delaying stack pushes as "dreaming", an appropriate activity for lazy computers. We wish to thank Robin Stanton, Richard Stallman, Jonathan Rees, and Richard Zippel for being the first readers of this paper and for making important suggestions, and Phil Agre for finding a bug.

References

[Actors]
Hewitt, Carl. "Viewing Control Structures as Patterns of Passing Messages." AI Journal 8, 3 (June 1977), 323-364.

[Bobrow and Wegbreit]
Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." Comm. ACM 16, 10 (October 1973) pp. 591-603.

[Burroughs]
Hauck, E.A., and Dent, B.A. "Burroughs' B6500/B7500 Stack Mechanism." Proc. AFIPS Spring Joint Computer Conference Vol. 32 (1968), 245-251.

[CONNIVER]
McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER Reference Manual. AI Memo 295a. MIT AI Lab (Cambridge, January 1974).

[Interpreters]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). MIT AI Memo 453 (Cambridge, May 1978).

[LISP Machine]
The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. LISP Machine Progress Report. AI Memo 444. MIT AI Lab (Cambridge, August 1977).

[Moses]
Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Revised Report]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME: A Dialect of LISP. MIT AI Memo 452 (Cambridge, January 1978).

[SCHEME]
Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).

[SCHEME Chip 0]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. "Storage Management in a LISP-Based Processor." Proc. Caltech Conference on Very Large Scale Integration (Pasadena, January 1979).

[SCHEME Chip 1]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. Design of LISP-Based Processors; or, SCHEME: A Dielectric LISP; or, Finite Memories Considered Harmful; or, LAMBDA: The Ultimate Opcode. AI memo 514. MIT AI Lab (Cambridge, March 1979).

[SCHEME Chip 2]
Holloway, Jack; Steele, Guy L., Jr.; Sussman, Gerald Jay; and Bell, Alan. The SCHEME-79 Chip. AI memo 559. MIT AI Lab (Cambridge, December 1979).

[SLIP]
Weizenbaum, J. "Symmetric list processor." Comm. ACM 6, 10 (September 1963), 524-544.

[Smalltalk]
Goldberg, Adele, and Kay, Alan. Smalltalk-72 Instruction Manual. Learning Research Group, Xerox Palo Alto Research Center (March 1976).

[Weizenbaum]
Weizenbaum, J. "Knotted list structures." Comm. ACM 5, 3 (March 1962), 161-165.